# Neural Transfer: Creating Art Using Deep Learning Techniques

Naimish Patel
*ANLY 535-50- A-2019*
*Harrisburg University of Science and Technology*
npatel22@my.harrisburgu.edu

Siddhant Mohapatra
*ANLY 535-50- A-2019*
*Harrisburg University of Science and Technology*
smohapatra@my.harrisburgu.edu

Neeraj Jayesh Bhatt
*ANLY 535-50- A-2019*
*Harrisburg University of Science and Technology*
nbhatt@my.harrisburgu.edu

Nirmesh Gollamandala
*ANLY 535-50- A-2019*
*Harrisburg University of Science and Technology*
ngollamandala@my.harrisburgu.edu

## I. INTRODUCTION

Neural Style Transfer was a concept introduced in the research paper published by Leon Gatys, Alexander Ecker and Matthias Bethge in September 2015[1]. Neural Style Transfer is a technique that uses 2 images and blends them together to get a transformed image on the base input image. It takes a content image, style reference image and a base input image that is to be styled and combines it and transforms the input image to reflect it like the content image with the style of the style image.

Figure 1: A simple example of this would be getting the background of a Picasso painting on Mona Lisa.

### *Rationale*

Neural Style Transfer is based on the principle to establish 2 distance functions. One function details out the distance between the content image and the generated image and the other function calls out the distances between the style image and the generated image. Both the functions are referenced as Content and Style. Now, once the 2 images: Content Image, Style Image are provided, the input image is transformed to ensure that the two distances described above are minimized.

The input image is transformed by minimizing the distance losses using backpropagation, and ultimately the final image that matches content of content image and style of the style image.

### *Practical Implementation*

Data Augmentation is currently one of the major areas that utilizes the Neural Style Transfer. To explain this further, there are applications or research that is being done, that requires a lot of training images. Getting training images can be time consuming and at the same be expensive. In order to avoid or mitigate this problem, one can leverage neural style transfer where the existing training images can be used with different style backgrounds and can be used as the training sample. This can be extrapolated to the e-commerce or fashion industry, where one can use someone's face as an input image and fit it different backgrounds. This has proven to result in a higher robustness in the trained CNN models and resulted in a much higher accuracy compared to training without style transfer augmentation[2].

The neural style transfer can create a new image with a different style and still preserve the features of the content image. A similar approach can be used on audio spectrograms to transform sound into music essentially a singing voice to a musical instrument[3][4].

Also, from an artistic standpoint, one can apply different styles to different images, and this has been one of key selling points of Neural Style Transfer.

## II. RELATED WORK

Currently, the implementation of Neural Style transfer has been extended to real time processing of videos, where each frame is taken and applied a varied background[5]. There has been work going on to improve the processing speed for images using a technique known as knowledge distillation where you leverage the learning of the main teacher network and use smaller student sub-networks for individual images. This significantly reduces training time[5].

Style Transfer is also being used to create iOS apps that can be used to transform photos and videos from phone into beautiful artistic images[6].

## III. DATA

### *Data description*

NST doesn't require a dataset but rather it depends upon the pre-trained convnet architecture to extract features from content and style images. As part of our project, we leveraged some custom images and varied backgrounds.

### *Challenges*

NST is based on perception, that implies that one cannot certify the accuracy of the work that is being done. From a technical standpoint, one can ensure that the inputs image that is being built on the content and style image is transformed by minimizing the distances losses.

Another big challenge is the computational complexity. Even though we need only two images there are plenty of parameters to train in the backpropagation. For example a conv layer of size 256 with 64 channels when unrolled into a matrix of 2 dimensions will have height times width no of rows and columns equal to the number of channels, which in this case 256*256*64 parameters for a single image. Hence training on a gpu is recommended to speed up the process and adjust the parameters to be addressed later in the report to run the training multiple times.

Hence we trained the network on google colab.

## IV. TECHNICAL APPROACH

### *Model*

Our goal is to paint a picture using the style image, the style of the image will then be transferred to the content image. We will obtain activation values from both content and style images. We will first feed the content and style image to a feed-forward CNN network and capture its activation values at the layer of interest. For the content cost we will compute the distance between the feature maps of the content image and the generated image. For the content cost we will use only one layer.

Our first approach is to use a pre-trained model rather than building a model from scratch. This is called Transfer Learning. We will be using the VGG19 model. This pre-trained model will be loaded and fed into the input tensor of our model. This will enable us to extract the features of the content and style images. As part of our analysis, we are going to discard the fully connected layers. We used 16 CONV layers, 5 pool layers of the 19 VGG-network. We haven't used the Fully Connected Layers as we are not focused on classifying. We ran this for 10 epochs, as it was computationally time-consuming.

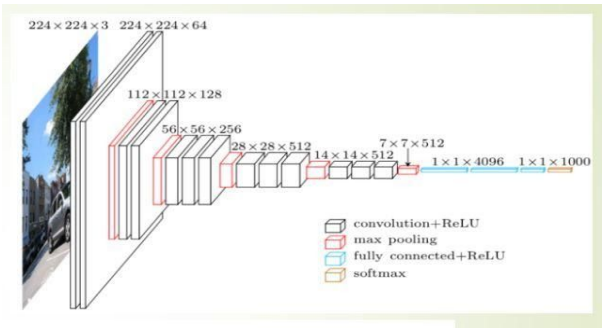


Figure 2: VGG19 Network

### Preprocessing

As part of our analysis, we have defined some image constraints. We have hardcoded the image width and height to be 500. Some images like the ones we used have 4 channels RGBA where A is alpha the 4th channel. However, VGG network expects 3 channels and hence we first convert the image from RGBA to RGB. Next step is mean normalization. For mean normalization we used values 123.68, 116.779 & 103.939 as these are the mean values of the training pixels for the ImageNet Classification on which networks like VGG19 were trained[7]. However, there is a reason we keep it in this order as 103.939 is the mean for the red channel, 116.779 is the mean for the green channel and 123.68 is for the blue channel but the images are converted to BGR. This is essential as the VGG19 model was trained using caffe that relies on openCV functionalities, which require images in BGR format.

Content and Style Images:

Figure 3: Neural Style Transfer Content Image

Figure 4: Neural Style Transfer Style Image

As mentioned earlier, the goal is to combine the neural content image with the style image to get a final Generated image.

Figure 5: Final Generated Image

***Neural Style Transfer Cost Function***

**Content Cost Function:**

This is nothing but Euclidian distance between content image and generated image. To explain this further, suppose we take one CONV layer, where we have heights and weights along with the number of channels, in order to come up with the Cost Function, we have to open it up and our heights and weights end up being the columns and channel becomes the row of our matrix.

This is then taken along with the generated image, to get the sum of square distance, which is being minimized by gradient descent.

$$J_{content}(C,G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{\text{all entries}} (a^{(C)} - a^{(G)})^2$$

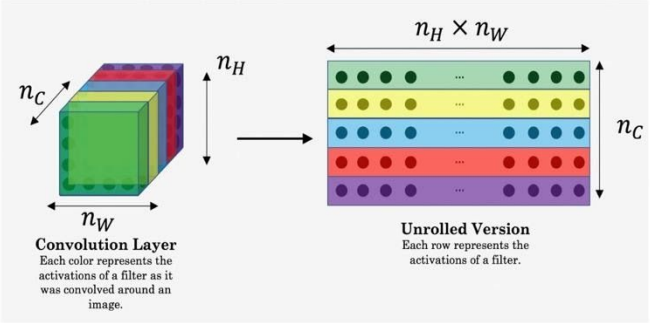


Figure 6: Content Cost Function

**Style cost function:**

We first compute the gram matrices of the style image and the generated image. To explain this further, just like Content Cost Function, in Style Cost Function, the weights, heights and channels are unrolled. But in addition to that, we take a dot product of the matrix and its transpose to get the Gram Matrix. This will then get computed along with the generated

image. The main reason to do this is to capture the cross-correlations between the channels of the styling image.

$$J^{[l]}_{style}(S,G) = \frac{1}{4 \times {n_C}^2 \times (n_H \times n_W)^2} \sum_{i=1}^{n_C} \sum_{j=1}^{n_C} (G^{(S)}_{(gram)i,j} - G^{(G)}_{(gram)i,j})^2$$

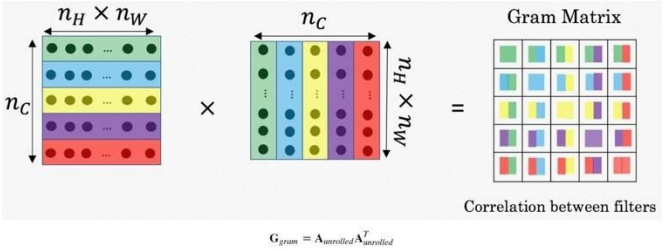


Figure 7: Style Cost Function

To further optimize the process the styling correlations are captured from multiple convolutional layers and then summed.

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{style}(S, G)$$

Here the lambda parameter decides how much weight to assign to each convolutional layer selected. We assigned equal weight to each layer.

Total Cost Function: Taking two parameters which are multiplied to the Content Cost Function and Style Cost Function. We initialized the alpha and beta with random values and then modified it based on the outputs we saw. This was run across 10 epochs.

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G)$$

At this stage we got the following result:

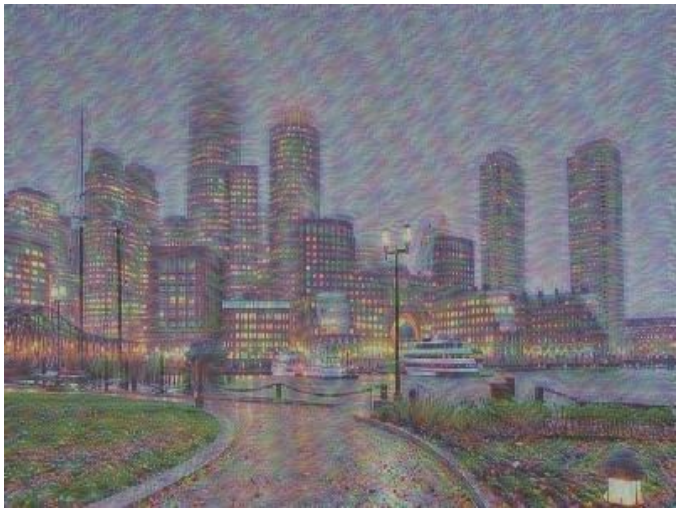

Figure 8: Initial generated output

We see one major issue. There is too much variation in the pixel values resulting in significant variation in the content image and style image resulting in improper immersion of the style image into the content image.

**Total Variation Regularizer:**

To control the variation between neighboring pixels we added a total variation regularizer that penalizes the variation between neighboring pixels of the output image and ensures spatial continuity.

The equation of the regulatizer is:

$$R_{V_\beta}(\mathbf{y}) = \lambda_{V_\beta} \sum_{i,j,k} \left( \left(y_{i,j+1,k} - y_{i,j,k}\right)^2 + \left(y_{i+1,j,k} - y_{i,j,k}\right)^2 \right)^{\beta}$$

This is simply the euclidean distance between each pixel height pixel and its previous height pixel plus the euclidean distance between each width pixel and its preceding width pixel for each channel of the output image. For e.g. let's say A is a 4 by 4 matrix . So essentially this function is (A[0,1] - A[0,0])^2 + (A[1,0] - A[0,0]). This is done across all pixels of all channels and then summed returning a scalar value. We also have an additional value **β** that represents the total variation loss factor. Essentially the scalar value returned by the sum of squared distances is raised to the power of **β**. This is then multiplied by a variation loss weight lambda parameter and then added to the content and style losses. Coupled with lambda **β** provides us greater control over the variation loss so that it doesn't overshoot.

So now total cost function:

$$J(G) = \alpha J_{content}(C,G) + \beta J_{style}(S,G) + Rv_{\beta}(G)$$

This resulted in a massive improvement in the generated image, which our final output above demonstrates.

To optimize this instead of using adam optimizer we used L-BFGS algorithm. This algorithm is a quasi-newton method to find the point at which the derivative of the objective function is 0. The Newton method uses the inverse hessian (second derivative), which is computationally expensive. For n parameters there are n by n second order derivatives. BFGS uses an approximation of the inverse hessian that is easier to compute. However, with too many parameters BFGS approximations can increase space complexity significantly.

L-BFGS does not store all approximations but only stores up to m vectors from which we can approximately reconstruct the matrix operations that would have been calculated if we had the full BFGS matrix.

L-BFGS works better than adam optimization when there are too many parameters and a smaller dataset especially if the parameter space has plenty of flat valleys. Here with no of parameters we had to optimize and only 2 images as inputs L-BFGS was a better option compared to Adam.

**Google Deep Dream:**

Google Deep Dream algorithm has practical implementation when it comes to visual graphics and video processing. Google created a musical video with hallucinating image effects using the deep dream algorithm[8]. The concept of Deep Dream can be related to a beautiful question, "What happens if Computers can dream or hallucinate?"

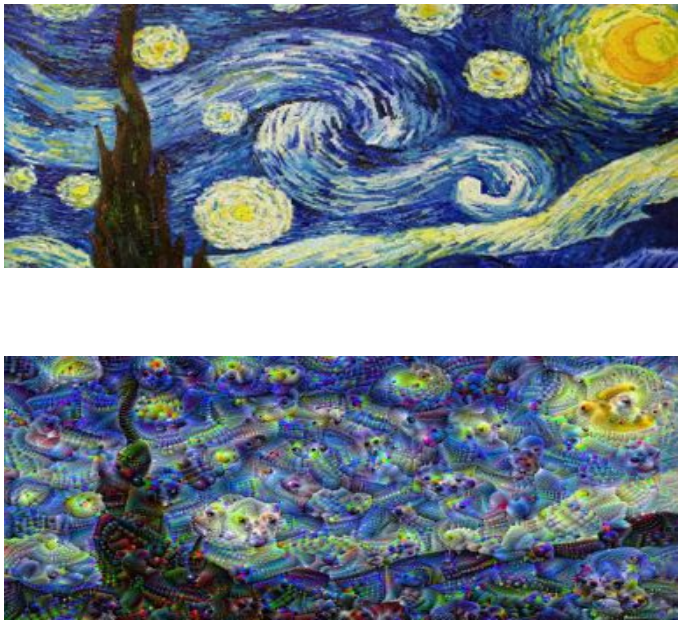

Figure 9: Pre and Post 200 Epochs, this is basically what images goes through within the CONV layers.

In the case of Google Dream Algorithm, it focuses on the middle CONV layers to understand what transformation an image goes through.

Google Dream Algorithm is based on an Inception network that focused on concatenating several middle layers into one single feature detector. These are known as concatenation layers. We will use the middle concatenation layers to shoot the activations to see hallucinating effects

Deep Dream uses convolutional neural network to create dreamlike hallucinogenic images. It deliberately over processes the image. It uses gradient ascent so new image will be more hallucinated than the previous layer. Deep Dream can be used to create a virtual environment which mimics the experience of psychoactive substances.

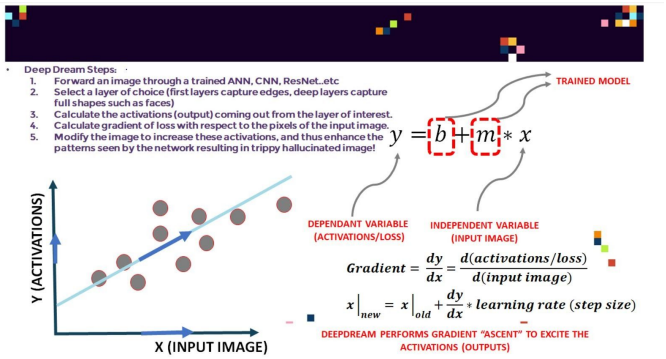


Figure 10: mathematical representation of deep dream

Here we can think of an activation and input image as a linear regression whose slope or gradient we are computing. This will tell us the change in activations to the change in input image. We will then add this gradient to the input image multiplied by a learning rate to maximize the loss in order to shoot up the activations. This in a way also tells us what these middle layers of the inception network see.

Following are steps of Deep Dream:

- Forward image through trained CNN Network.
- Select layer of choice (first layer captures edge, other layers capture full shape)
- Extract the activations coming out from each layer
- Calculate gradients of the activations with respect to pixels of the input image
- Modify image to increase these activations. Thus enhancement in pattern seen by network will result into hallucinated image.

## V. REFERENCES


1. Gatys, Leon A., et al. "A Neural Algorithm of Artistic Style." *Journal of Vision*, vol. 16, no. 12, 2016, p. 326., doi:10.1167/16.12.326.
2. Jackson, Philip T., et al. "Style Augmentation: Data Augmentation via Style Randomization." *ArXiv.org*, 12 Apr. 2019, arxiv.org/abs/1809.05375.
3. Verma, Prateek, and Julius O. Smith. *Neural Style Transfer for Audio Spectrograms*. nips2017creativity.github.io/doc/Neural_Style_Spectograms.pdf.
4. Choksi, Bhaumik, et al. "Style Transfer for Audio Using Convolutional Neural Networks."

*International Journal of Computer Applications*, vol. 175, no. 8, 2017, pp. 17–20., doi:10.5120/ijca2017915612.

5. Baumli, Kate. “Real Time Video Neural Style Transfer.” *Medium*, Towards Data Science, 15 Dec. 2018, towardsdatascience.com/real-time-video-neural-style-transfer-9f6f84590832.
6. Omar M’Haimdat. “Style Transfer on IOS Using Convolutional Neural Networks.” *Medium*, Heartbeat, 15 Oct. 2019, heartbeat.fritz.ai/style-transfer-on-ios-using-convolutional-neural-networks-616fd748ece4.
7. Pattanayak, Santanu. “Intelligent Projects Using Python.” *Google Books*, Google, books.google.com/books?id=mGSGDwAAQBAJ&pg=PA54&lpg=PA54&dq=why%2Bdo%2Bwe%2Buse%2B123.68%2C%2B116.779%2C%2B103.939%2Bfor%2Bmean%2Bnormalization&source=bl&ots=-wtZ3iihUK&sig=ACfU3U1rtYqLB6Qsbi9r9osjHXBaeHGW5g&hl=en&sa=X&ved=2ahUKEwif0-_y--DnAhVGnOAKHfnKAHAQ6AEwDHoECAkQAQ#v=onepage&q=why%20do%20we%20use%20123.68%2C%20116.779%2C%20103.939%20for%20mean%20normalization&f=false.
8. “DeepDream, the First Music Video Created with Google Neural Network.” *LifeGate*, Google, 22 Mar. 2018, www.lifegate.com/people/lifestyle/deepdream-video-music-google-neural-network.